Streams, Tuples, Unions, and Comprehension

CS395T

Fall 2003

From Meijer and Schulte

- Impendence mismatch for mid-tier in Web App
  - Middle tier Java or C# software in n-tier Web Application vs Databases
- Need for a growing language
  - Guy Steele: the language must grow …
- Result: modern OOL + tables & documents
  - Proposed data types: Streams, Tuples, Unions, Content Classes, Queries

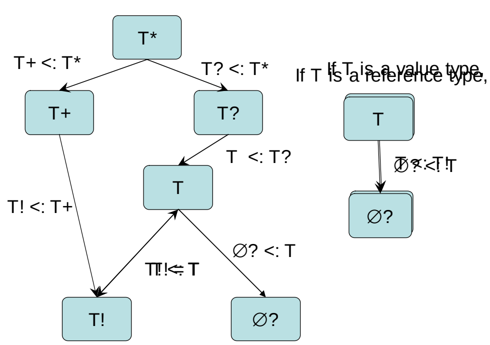
Streams

- An Improvement to IEnumerable
- * arbitrary length of homogenous data
- * (≥0 elements), + (>0 elements), ! (=1 element), ? (≤1 element)
- yield for returning a stream

Streams (Cont'd)

- null for empty Streams
- Covariance
  - If S <: T, then S* <: T*
- Flattening
  - T?! = T!, T*+ = T+

Streams: Type Hierarchy



Tuples

- Sequences of heterogeneous data
- Values are labeled or unlabeled
- Contrast with
  - records (labeled and unordered)
  - regular tuples (unlabeled and ordered)
  - advantage?
- Example

```
sequence{Button b; string;}
```

  - First value is labeled b; second value is unlabeled

Streams+Tuple

- Build Tables from Streams and Tuples

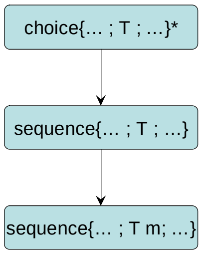
```
enum FiveWalk {Metal, Wood, Water, Fire, Earth}
enum Year {Rat, Ox, Tiger, Rabbit, Dragon, Snake,
  Horse, Goat, Monkey, Rooster, Dog, Boar}
Type FengShui = sequence{
  string Name; FiveWalk element; Year animal; int*
  badMonths;
}
FengShui* illogical;
```


Unions

- Variants
- Choices are idempotent, associative and commutative
- Often used as a member
- Example

```
Class Address{
  choice{string Street; int POBox;}
  string City;
}
```

Tuples: Type Hierarchy



sequence{string name, year animal}
can be upcast to
choice{string name, year animal}*

sequence{“John”, Rabbit} upcast to
[“John”, Rabbit]

Content Classes

- Uses XML for class declaration
- Intuitive correspondence between XSD particles and previously proposed types

```
<element name="FengShui">
  <complexType>
      <element name="Name" type="string"/>
      <element name="element" type="FiveWalk"/>
      <element name="animal" type="Year"/>
      <sequence>
            <element name="badMonths" type="int"/>
      <sequence/>
  <complexType/>
<element/>
```


Data Access

- Wildcard, Transitive and Type-based Member-access
  - Wildcard * returns a stream of all members.
  - Transitive .. is used as a path access; type qualifier chooses members of a certain type
- Select and Join operations
  - Optimization? Both operations act on streams which are stateful?
- Map, Filter and Fold (Apply-to-all block)

Data Access: Map and Fold

- Comprehension-like features

```
int* nats = {int i=0; while(true) yield i++;};
Haskell equivalent: [i | i <- [1..]]


int mapsum(int s, int* xs){
  xs.{s += it; return;}; return s;}
```

- Map :: $(a \rightarrow b) \rightarrow \{a\} \rightarrow \{b\}$
- Fold :: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow \{b\} \rightarrow a$

Comprehension Syntax

- Apply List Comprehension for Queries.
- Structural Recursion allows recursive functions/queries written in pattern matching style.

List Comprehension

```
[1..4] = [1, 2, 3, 4]

[1..] = Infinite List 1, 2, 3,… with lazy evaluation

[(x,y) | x <- [1..3], y <- [1..3]] =
  [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),
   (3,1),(3,2),(3,3)]

sort [] = []
sort (x:xs) = sort [u|u<-[xs],u<=x] ++ [x] ++
              sort [u|u<-[xs],u>x]
```


Query Comprehension

```
{[Name = p.Name, Mgr = d.Mgr] |
  \p <- Emp,
  \d <- Dept,
  p.DNum = d.DNum}
```

- For every Emp
  For every Dept
  only elements whose DNum’s are the same

Query Comprehension (Cont’d)

- Collections: Bag {| |}, List {|| ||}, Set { }
- Top level declarations: define
- Relational algebra

```
define join(\x,\y) =>
    {[A = u, B = u’, D = v’] |
     [A = \u, B = \u’] <- x,
     [C = u’, D = \v’] <- y}
```

Query Comprehension (Cont'd)

- Structural Recursion

```
define  fibonacci(0) => 1
        | fibonacci(1) => 1
        | fibonacci(n) =>  fibonacci(n-1) +
                           fibonacci(n-2)
```

- Use ext for comprehension

```
ext(f) = define h({| |}) => {| |}
                | h({|\x|}) => f(x)
                | h(\s1 ⊕ \s2) => h(s1) ⊕ h(s2)
```


Power of Comprehension

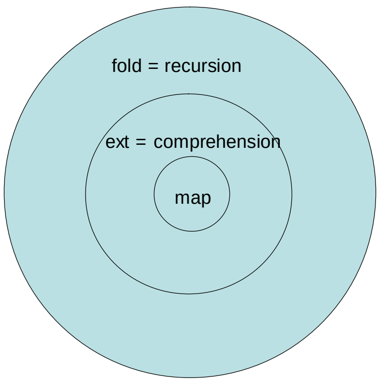
```
1.  {e | \x <- S,G} = ext(f)S where f(\x) = {e|G}
2.  {e | C,G} = if C then {E|G} else { }
3.  {e | } = {e}
```

map :: (a → b) → {a} → {b}

ext :: (a → {b}) → {a} → {b}

map f x = ext g x where g a = {f a}

Power of Comprehension

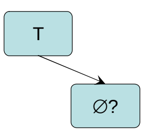


Conclusion

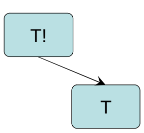
- Meijer and Schulte put together lots of interesting ideas: stream, tuple, etc. data types, XML and comprehension.
- Comprehension is an very appropriate and elegant way for relational query construction. (Personal preference)

Streams Type Hierarchy (Cont'd)

If T is a reference type, null type is a subtype of T.



If T is a value type, T is a subtype of T!.




```
e = [A = u, B = u', D = v']
\x = [A = \u, B = \u']
S = m
G = [C = u', D = \v'] <- n
join(\m, \n) =
{[A = u, B = u', D = v'] | [A = \u, B = \u'] <- m,
 [C = u', D = \v'] <- n} =
ext(f)m where
  f([A = \u, B = \u']) =
  {[A = u, B = u', D = v'] | [C = u', D = \v'] <- n} =
  ext(g)n where
      g([C = u', D = \v']) =
      {[A = u, B = u', D = v'] | } =
      {[A = u, B = u', D = v']}
```